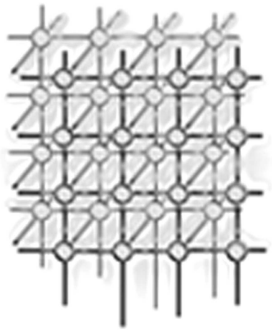
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Self-Maintained Distributed Tuples for Field-based Coordination in Dynamic Networks‡



Marco Mamei*,†, Franco Zambonelli†

Dipartimento di Scienze e Metodi dell'Ingegneria, Università di Modena e Reggio Emilia, Viale Allegri, 13, 42100, Reggio Emilia, Italy.

SUMMARY

Field-based coordination is a promising approach to orchestrate the activities of components in a wide range of application scenarios. To implement such an approach, one can rely on distributed tuples injected in a network and then propagated to form field-like distributed data structures to be sensed by application components. Moreover, to gain the full benefits from such an approach, it is important to enable the distributed tuples to preserve their structures despite the dynamics of the network. In this paper, we show how a variety of self-maintained distributed tuples for field-based coordination can be easily programmed in the TOTA middleware. Several examples clarify the approach, and a case study is detailed throughout the paper to ground the discussion. Eventually, performance data is presented to verify the effectiveness of the approach.

KEY WORDS: Field-based coordination; Field-like distributed data structures; tuple spaces; dynamic networks; mobility; programming.

1. INTRODUCTION

In the near future, computer-based systems will be embedded in all our everyday objects and in our everyday environments. These systems will be typically communication enabled, and capable of coordinating with each other in the context of complex, mobile and distributed applications. The operational environment where such applications will take place will be

*Correspondence to: Marco Mamei, Dipartimento di Scienze e Metodi dell'Ingegneria, Università di Modena e Reggio Emilia, Viale Allegri, 13, 42100, Reggio Emilia, Italy.
†E-mail: {mamei.marco, zambonelli.franco} @unimo.it
‡This work was supported by Italy's Ministry for Higher Education, Training, and Research and its National Research Council in the "Progetto Strategico IS-MANET: Infrastructures for Mobile Ad-Hoc Networks" project

Copyright © 2000 John Wiley & Sons, Ltd.
Received 10 May 2004
Revised XX September 200X

extremely dynamic. On the one hand, application "agents" (we use this term generically to indicate the active components of a distributed application) will be likely to depart out of and arrive into the network at any time, and to roam across different networks and environments. On the other hand, the very structure of the network will be open and dynamic with nodes and resources varying suddenly.

Coordinating agents' activities in such dynamic network environments with traditional models and infrastructures is not easy. It is of great importance to rely on models and middleware infrastructures explicitly designed to take care of all the issues implied in network dynamics. From our perspective, one of the basic problem in supporting agents' coordination is to provide them with an effective representation of their operational environment (simply *context* from now on). Such representation, in fact, defines what the agents *see* about their environment and thus implicitly conditions their behavior. In particular, such context representation must: *(i)* be simple and cheap to be obtained; *(ii)* be expressive enough to facilitate the definition and the execution of the agents' activities; *(iii)* continuously reflect the actual state of the dynamically changing context.

With this regard, approaches based on the concept of field-based coordination [4, 6] appear very suitable. In these approaches, the agents context is abstracted by means of abstract "fields" expressing forces driving agents' activities, resembling the way the gravitational field drives mass particles in our universe. In these approaches a middleware infrastructure is required to create and support the field representation. Moreover, to offer an up to date representation of the agent context, fields must adapt and self-maintain their value to reflect possibly changing environmental conditions. Agents can locally perceive these fields and decide what to do on the basis of the fields they sense and their magnitudes.

This field-based representation of the context is *(i)* simple and cheap to be obtained, since fields are spread in the environment and agents need only to perceive them locally; *(ii)* expressive enough to facilitate the coordination activities, since agents need only to sense fields and be driven by them. *(iii)* robust and adaptive to the above mentioned environmental dynamism due to fields self-maintenance.

From the programmer viewpoint, the main task in this approach is to map specific coordination problems into suitable field-based representations. To this end, a programming model specifying how to build fields is required. The contribution of this paper is to present a model to program self-maintained distributed tuples, implementing the concept of fields.

This paper is organized as follows: Section 2 introduces field-based coordination and presents "Tuples On The Air" (TOTA); a middleware conceived to support fields by means of self-maintained distributed tuples. Section 3 details how to program distributed tuples in TOTA. Section 4 presents performances and experiments. Section 5 concludes.

2. FIELD-BASED COORDINATION

To realize the idea of field-based coordination we basically need: *(i)* to represent fields by means of suitable data structures; *(ii)* an infrastructure holding and supporting these data structures. Field-like data structures can be either thought as pre-existing in the infrastructure (i.e. created at bootstrap and possibly later affected by agent actions) or injected by the agents at run time

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

and being maintained by the infrastructure further on. Given that, field-based coordination is centered on a few key concepts:

1. The agents' operational environment is represented by "field-like data structures", created by agents and/or by the infrastructure, and locally sensed by agents;
2. A coordination policy is realized by allowing the agents to locally sense field-like data structures stored in the infrastructure and act driven by them;
3. Both environment dynamics and agents' actions may induce the field-like data structures to update (i.e. self-maintain) their distributed values. This induces a feedback cycle (point 2) that can be exploited to globally achieve an adaptive coordination pattern.

2.1. Application Scenarios and Related Work

To clarify the ideas of field-based coordination and to show its applicability, let us focus on some application scenarios.

In robotics, the idea of fields driving robotic movement is not new [3]. For instance, one of the most recent manifestations of this idea, the Electric Field Approach [4], was used to control a team of Sony Aibo legged robots in the RoboCup domain. Following the EFA approach, each Aibo robot builds a field-based representation of the environment from the images captured by its head-mounted camera. *Good* positions to go are represented by means of attractive fields. *Bad* positions are represented by means of repelling fields. Each robot decides its movements by examining the gradients of such fields. Fields are updated to take into account changes in the environment. Similar ideas have been used in a variety of circumstances ranging from pervasive computing [6] to video-games [13].

An amorphous computer [10] consists of massive numbers of identically-programmed and locally-interacting computing agents, embedded in space. It can be modelled as a collection of "computational particles" sprinkled randomly on a surface or mixed throughout a volume. Field-like data structures can be spread in the amorphous computer to let various patterns and shapes emerge. Just to mention few trivial examples, if a leader particle creates a field that propagates increasing its magnitude until reaching a maximum value, one can create approximately circular regions of controlled size. If a line of particles propagate the above field, stripes instead of circles can be identified in the amorphous computer [10].

A modular or self-reconfigurable robot is a collection of simple autonomous actuators with few degrees of freedom connected with each other. A distributed control algorithm is executed by all the actuators to let the robot assume a global coherent shape or a global coherent motion pattern (i.e. gait). Some proposed approaches [12] adopt field-like data structures to control such a robot. Specifically, a distributed shape or motion gait is encoded by means of fields spread across the robot specifying how the robot's actuators should bend.

The behavior of ants, as social insects, has recently inspired several algorithms used to control the activities of multi agent systems [1]. The key to these approaches is in emulating the way in which ants interact one another. They do so by means of pheromone signals they spread in the environment that will be perceived by other ants later on. These pheromone signals can be used to find food sources, or to coordinate efforts in moving some heavy objects, etc. Pheromone signals can be easily modelled by means of fields driving agents activities [5].

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

Instead of being propagated by the infrastructure in all directions, field-like data structures implementing the concept of pheromone would be distributed by the agents themselves as they move across the network. These data structures would then be used as trails driving agents' activities. The research projects Anthill [8] and SwarmLinda [9] share the idea of applying ant-inspired algorithms to Internet-scale Peer-to-Peer systems. Here, field-like data structures - modelling ants' pheromones - create paths connecting peers that share similar files, thus enabling, for example, an effective content-based navigation in the network of peers.

Routing in Mobile Ad-Hoc Network (MANET) can be easily modelled as a coordination problem: agents (i.e. network nodes) need to cooperate forwarding each other messages to enable long-range, multi-hop communication. The main principle underlying many routing algorithms is to build a distributed data structure (implemented by means of a set of distributed routing tables) suitable to provide route information. Specifically, these data structures create paths in the network enabling agents to forward messages in the right direction. These paths (i.e. data structures) are maintained to take into account changes in the network topology. Although a few routing protocols make the analogy with fields explicit [11], the idea at the basis of distributed routing data structure is the same as field based coordination: provide agents with a ready-to-use representation of the context (i.e. where the message should go next).

We think that there are two main drawbacks in almost all these related works. First, fields are actively spread and maintained by the agents. On the contrary, we think that enabling a field to self-maintain is a fundamental property from a software engineering perspective. In this way, fields become a *fire and forget* mechanism: agents have simply to inject them without further taking care of their update. Second, a general-purpose programming model, enabling to express different kinds of fields, is missing in almost all the related works. We think that such a programming model is a fundamental requirement to apply the field-based approach to a vast array of scenarios in a uniform way.

2.2. Follow-through Case Study Scenario

A central issue in the development of any kind of distributed application is to provide components (i.e. agents) with suitable and effective mechanisms to access distributed information. In its broader meaning, information access is at the basis of data sharing, context-awareness (i.e. access to context-related information) and also interactions (i.e. access to communication partners).

Location-based mechanisms to access distributed information enable an agent to access resources within suitable locality constraints (e.g. find all the printers on this floor, or find the closest gas station). In general, this kind of mechanism can be very useful in a wide range of application scenarios. For example, in a B2C (business to consumer) scenario, a customer may want to limit the scope of a query for a specific item to the city where (s)he lives, to save shipping cost and time. Considering mobile computing scenarios, location-based access to information and services becomes even more important [2]. Most mobile computing applications become really useful when it is possible to limit the scope of queries to those resources and services actually reachable by the mobile users (e.g. mobile users ask their PDA to retrieve nearest restaurants while roaming through a city). When even the resources and the

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

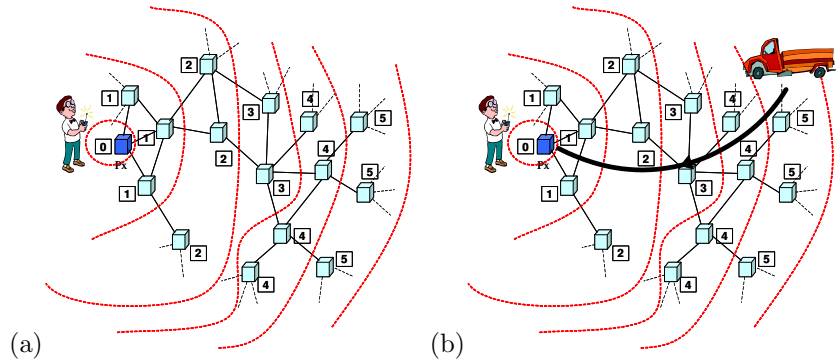


Figure 1. (a) An agent propagating a QUERY field to look for a specific information. (b) A suitable bus propagates an ANSWER field that propagates following downhill the QUERY field.

services are mobile, location-based access enables new and exciting applications. For example, users may want to access the nearest bus going to a specific direction and arrange a "rendez-vous" at a specific bus-stop. This kind of system could greatly improve the flexibility of the bus service (e.g. by taking into account possible delays, arranging on-the-fly stops, etc.). Similar considerations apply to a variety of other scenarios: booking the nearest free taxi, on-the-fly car sharing and car pooling, etc.

In general, we think that field-based concepts are very useful in this scenario. In fact, fields spread in an environment can easily define regions that can be used to constrain the scope of the information access. As a simple example, we can envision that an agent looking for some information will create and spread across the mobile network a QUERY field carrying on the information about what the agent is looking for (e.g. bus) and that propagates the field within a specified locality scope. In more detail, the QUERY field can be flooded across the network infrastructure, but stopping propagating once a specified critical distance (e.g. 300m) from the source has been reached. Moreover, the QUERY field, by incrementing one of its values by one at every propagation hop, can create a routing structure to collect back the answers. Peers can look for incoming QUERY fields and possibly answer by propagating an ANSWER field carrying suitable information (e.g. Bus N. 21, driving to Broad st.). The ANSWER field would propagate following the QUERY field's routing structure downhill to reach the enquiring agent without flooding the network (see figure 1). It is worth noting that fields are well-suited to such highly mobile scenarios since they automatically update to reflect changes in the environment situation. Thus, ANSWER fields are able to route back to the enquiring agent, even if the bus or the agent moves after issuing the QUERY field.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

2.3. The TOTA Middleware

The idea of fields can potentially be implemented on any distributed middleware providing basic support for data storing (to store field values), communication mechanisms (to propagate fields) and event-notification mechanisms (to update fields and notify agents about changes in fields' values). However, we think that a set of distributed tuple spaces is a particularly fertile ground on which to build such idea. A set of logically bounded tuples, one in each of the distributed tuple spaces, naturally matches the idea of a field spread across the infrastructure. The content of a tuple would represent the field's physical properties (e.g. magnitude). This is the approach taken by our middleware TOTA (Tuples On The Air) [7].

TOTA is composed of a peer-to-peer network of possibly mobile nodes, each running a local version of the TOTA middleware. Upon the distributed space identified by the dynamic network of TOTA nodes, each component is capable of locally storing tuples and letting them diffuse across the network. Tuples are injected in the system from a particular node, and spread hop-by-hop accordingly to a specified propagation rule. Specifically, in TOTA, fields have been realized by means of distributed tuples **T=(C,P)**, characterized by a content **C** and a propagation rule **P**. The content **C** is an ordered set of typed elements representing the information carried on by the tuple. The propagation rule **P** determines how the tuple should be distributed and propagated in the network. This includes determining both how a tuple's content should change while it is propagated and the "scope" of the tuple (i.e. the distance at which such tuple should be propagated and how such propagation can be affected by the presence or the absence of other tuples in the system). In addition, the spatial structures induced by tuple propagation must be maintained coherent despite network dynamism. For instance, when new nodes get in touch with a network, TOTA automatically checks the propagation rules of the already stored tuples and eventually propagates the tuples to the new nodes. Similarly, when the topology changes due to nodes' movements, the distributed tuple structure automatically changes to reflect the new topology. The main contribution of this article, in the following sections, is to actually show how to *program* the propagation rule **P** of tuples to let them distribute and self-maintain accordingly to specified patterns.

Application agents and tuples can access the TOTA middleware via a proper API. The TOTA API provides functionalities to let the application to inject and delete tuples in the local middleware (*inject* and *delete* methods), to read tuples both from the local tuple space and from the node's one-hop neighborhood, either via pattern matching or via key-access to a tuple's unique id (*read*, *readOneHop*, *keyrd*, *keyrdOneHop*). Moreover, TOTA supports reactive behaviors by allowing an agent to *subscribe* and *unsubscribe* to relevant events (e.g. the arrival of a new tuple) and to call-back the agent's *react* method whenever relevant events happen. It is worth noting that, despite the fact that all the TOTA read methods are non-blocking, it is very easy to realize blocking operations using the event-based interface. An agent willing to perform a blocking read, has simply to subscribe to a specific tuple and wait until the corresponding reaction is triggered to resume its execution.

Finally, two methods (*store*, *move*) allow tuples to be actually stored in the local tuple space and to migrate to neighboring nodes. These methods can be used only within the tuples' code. It is important to remark the difference between the *inject* and the *store* methods: *inject* is called by an agent to insert a new tuple in the TOTA network. Once injected, a tuple

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

autonomously travels hop-by-hop across the network (calling the *move* method). *store* is a low-level method called by a tuple to be archived in the local tuple space of the TOTA node currently being visited. If a tuple does not call the *store*, it simply propagates across the network without leaving any traces of its travel. Further details of the tuple life-cycle follow later on.

From the agent's perspective, executing and interacting basically reduces to inject tuples, e.g. *tota.inject(new QueryTuple());* and acting on the basis of the local tuples, e.g. *t = tota.read(new AnswerTuple()); if (t !=null) doSomeAction();*

3. PROGRAMMING DISTRIBUTED TUPLES

Other than the TOTA API, a suitable programming model is required to build TOTA distributed tuples. Within TOTA, distributed tuples have been designed by means of objects: the object state models the tuple content, while the tuple's propagation has been encoded by means of a specific propagate method. Following this schema, an abstract class *TotaTuple* has been defined to provide a general framework for tuples programming (see figure 2).

In TOTA, a tuple does not own a thread, but it is actually executed by the middleware that runs the tuple's *init* and *propagate* methods. Tuples, however, must remain active even after the middleware has run their code. This is fundamental because their self-maintenance algorithm, for example, must be executed whenever the right condition appears (e.g. when a new peer connects to the network, the tuples must propagate to this newly arrived peer). To this end, tuples can place subscriptions as provided by the standard TOTA API. These subscriptions let the tuples remain "alive", being able to execute upon triggering conditions.

A programmer can obtain new tuples by subclassing the *TotaTuple* class. However, to facilitate this task we developed also a tuples' class hierarchy (see figure 3). Subclassing from the tuples in the hierarchy makes it easy to code tuples dealing with propagation and maintenance automatically with regard to a vast number of circumstances.

3.1. StructureTuple

The only child of the *TotaTuple* class is the class *StructureTuple*. This class is a template to create distributed data structures over the network. However, *StructureTuples* are still not self-maintained: if the topology of the network changes the tuple local values are left untouched.

This kind of tuple can be used in applications where the network infrastructure is relatively static and thus there is not the need to constantly update and maintain the tuples because of network dynamism. So, if we set the case study application in an ad-hoc network with fixed nodes, *StructureTuple* could well serve to implement QUERY and ANSWER fields.

The *StructureTuple* class inherits from *TotaTuple* and implements the superclass method *propagate* realizing a general-purpose propagation schema that is at the core of the whole tuples' hierarchy (see figure 4). In particular, this propagation method is realized by executing a sequence of other specific methods: *decideEnter*, *decidePropagate*, *changeTupleContent* and *makeSubscriptions* so as to realize a breadth first, expanding ring propagation. The result is simply a tuple that floods the network without changing its content:

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

```
abstract class TotaTuple
{
 /* each tuple will receive a reference to the TOTA middleware
 in which it is actually executing */
 protected TotaInterface tota;
 /* the object instance variables represent the tuple fields */
 ...
 /* this method inits the tuple, by giving a
 reference to the current TOTA middleware */
 public void init(TotaInterface tota) {
  this.tota = tota;
 }

 /* this method codes the tuple actual propagation actions */
 public abstract void propagate();

 /* this method enables the tuple to react to occurring events */
 public void react(String reaction, String event)
 { }}
```

Figure 2. The abstract class TOTA tuple

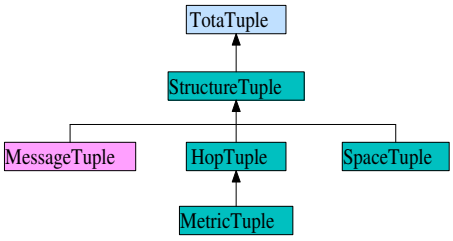


Figure 3. The TOTA tuples' class hierarchy

- When a tuple arrives in a node (either because it has been injected or it has been sent from a neighbor node) the middleware executes the *decideEnter* method that returns true if the tuple can enter the middleware and actually execute there, false otherwise. The standard implementation returns true if the middleware does not already contain that tuple.
- If the tuple is allowed to enter the method *decidePropagate* is run. It returns true if the tuple has to be further propagated, false otherwise. The standard implementation of this method returns always true, realizing a tuple's that floods the network being recursively propagated to all the peers.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

```
public final void propagate() {
 if(decideEnter()) {
  boolean prop = decidePropagate();
  changeTupleContent();
  this.makeSubscriptions();
  tota.store(this);
  if(prop)
   tota.move(this);
}}
```

Figure 4. The basic implementation of the propagate method

- The method *changeTupleContent* changes the content of the tuple. The standard implementation of this method does not change the tuple content.
- The method *makeSubscriptions* allows the tuple to place subscriptions in the TOTA middleware. As stated before, in this way the tuple can react to events even when they happen after the tuple completes its execution. The standard implementation does not subscribe to anything.
- After that, the tuple is inserted in the TOTA tuple space by executing *tota.store(this)*.
- Then, if the *decidePropagate* method returned true, the tuple is propagated to all the neighbors via the command *tota.move(this)*. The tuple will eventually reach neighboring nodes, where it will be executed again. It is worth noting that the tuple will arrive in the neighboring nodes with the content changed by the last run of the *changeTupleContent* method.

Programming a TOTA tuple to create a distributed data structure basically reduces to inherit from the above class and to overload the four methods specified above to customize the tuple behavior. Examples of tuple code will follow later on.

3.2. MessageTuple

MessageTuples are used to create messages that are not stored in the local tuple spaces, but just flow in the network as sorts of "events". The basic structure is the same as *StructureTuple*, but a default subscription is in charge to erase the tuple after some time passed. Note that it would not be possible to simply remove the *tota.store()* method from the *propagate* method, because previously stored values are used to block the tuple's backward propagation. To this end the tuple's value can be deleted only after the tuple "wave-front" has passed. It is worth noting that setting the time before deletion is not trivial. If the tuple propagates in a breadth first manner, it can simply be set to the time the tuple "wave-front" takes to proceed two-hops away. However, if the tuple is propagated in a specific direction and the network topology is closed in a circular track this can lead to a message that continues circulating through the network endlessly. For this reason, in such asymmetrical situations

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

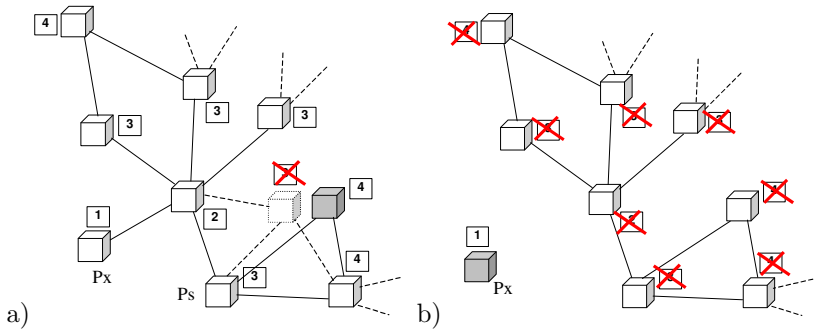


Figure 5. HopTuples self-maintain despite topology changes. (a) the tuple on the gray node must change its value to reflect the new hop-distance from the source Px. (b) if the source detaches all the tuples must auto-delete to reflect the new network situation.

MessageTuples must be used really carefully. Message tuples could be fruitfully applied as a communication mechanism. These tuples, in fact, could embed in their propagation rule the routing policy, without requiring routing agents to properly forward them. Moreover, it is easy to implement with these tuples several different communication patterns like unicast or multicast. Finally, by combining these tuples with *StructureTuples*, it is easy to realize publish-subscribe communication mechanism, in which *StructureTuples* create subscription paths, to be followed by *MessageTuples* implementing events. For instance, the ANSWER field in the case study could possibly be implemented with these tuples.

3.3. HopTuple

This kind of tuple inherits from *StructureTuple* to create distributed data structures that self-maintain their structure (supported by the TOTA middleware), to reflect changes in the network topology (see figure 5). Basically this class overloads the empty *makeSubscription* method of the *StructureTuple* class, to let these tuples react to changes in the topology, by adjusting their values to always be consistent with the hop-distance form the source (i.e. an integer hop is maintained in the *HopTuple* class).

These kinds of tuples are fundamental, since they enable field-based coordination also in presence of dynamic networks. The strength of these tuples, from a software engineering point of view, is that agents have simply to inject these tuples without further taking care of their update. All the burden in maintaining tuples is moved away form the agents.

Turning the attention to our case study, we can model QUERY fields by means of tuples propagating within a locality scope from where the agent is located. To this end we can think of creating the class *QueryTuple* that subclasses *HopTuple*, specifying a maximum number of hops the tuple is allowed to travel (see figure 6). It is worth noting that in an ad-hoc network scenario, hop-count is related to some extent to physical distance, especially if the network

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

```
public class QueryTuple extends HopTuple {
private int SCOPE = 3;
public boolean decidePropagate(){
  if(hop < SCOPE) return true;
  else return false;
}}
```

Figure 6. The tuple modelling the QUERY field. the hop counter is maintained by the superclass

is dense enough. An ANSWER field, instead, could be modeled by means of *AnswerTuple*: a tuple that follows the hop counter of a *QueryTuple* tuple downhill. To code this tuple one has basically to overload the *decideEnter* method to let the tuple enter only if the value of the *QueryTuple* in the node is less that the value on the node from which the tuple comes from (see figure 7).

3.4. MetricTuple and SpaceTuple

In some application scenarios, even in our case study, it would be helpful to ground tuple propagation to actual physical distances rather than to network distances (e.g. 3Km NORTH from the source, rather than 30 network hops from the source). To this end a common shared coordinate system must be established over the network. Relying on such a coordinate system, nodes are provided with a common knowledge of where e.g. the NORTH is and what is the physical distance between nodes.

Metric and *Space* tuples allow the creation of common shared coordinate systems across the TOTA network. In particular, both these tuples have three float numbers (X,Y,Z) as a content. Once one of these tuples is injected in the network, it propagates changing its content so that (X,Y,Z) reflect the coordinates of the node in a coordinate system centered where the tuple was first injected (see figure 8-a).

To support *Metric* and *Space* tuples, the TOTA nodes must be provided with some kind of localization device. The current implementation of these tuples requires the presence of either a GPS-like device or a RADAR-like device. The key difference between them is that: a GPS-like device provides absolute spatial information (e.g. latitude and longitude), a RADAR-like device provides local information (e.g. relative distances and orientations).

The implementation of *Metric* and *Space* tuples, given the availability of a GPS-like device, is straightforward. Once injected, the tuple will store the injecting node GPS coordinates and will initialize its content to (0,0,0). Upon reaching a new node, it will change the content to the GPS coordinates of the new node shifted back by the injecting node coordinates. Tuple update proceeds similarly: when a node moves (not the source one), the tuple locally changes its content by accessing the new GPS information.

The implementation of *Metric* and *Space* tuples, given the availability of a RADAR-like device, is more complicated. Here the goal is to create a tuple class that combines the local

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

```
public class AnswerTuple extends HopTuple {
 public String name;
 public int oldVal = 9999;
 QueryTuple trail;

 public boolean decideEnter() {
  super.decideEnter();
  int val = getGradientValue();
  if(val < oldVal) {
   oldVal = val;
   return true;
  }
  else return false;
 }
 /* this method returns the minimum hop-value of the
 QueryTuple tuples matching the tuple to be followed
 in the current node */
 private int getGradientValue() {
  Vector v = tota.read(trail);
  int min = 9999;
  for(int i=0; i<v.size(); i++) {
   QueryTuple gt = (QueryTuple)v.elementAt(i);
   if(min > gt.hop)
    min = gt.hop;
  }
 return min;
}}
```

Figure 7. The tuple modelling the ANSWER field

coordinate systems, built by the RADAR-like devices, into a shared coordinate system, with the center in the node that injected the tuple.

To explain how this can be achieved let us consider figure 8-b. The tuple (0,0,0) travels from P1 to P2 and it changes its content there. Specifically, it subtracts from its old value the coordinates of P1 as sensed by the RADAR-like device in P2. Thus $(0-(-100), 0-(-20), 0-0) = (100, 20, 0)$. It is worth noting that, in the figure, all the private coordinate systems are aligned. So combining them is just a matter of adding the coordinates. However, this perfect alignment is unlikely to happen and slightly more complex (geometric) combination will be required. Tuple update proceeds similarly: once these tuples have been propagated, if a node moves (not the source one), only its tuple local value is affected, while all the others are left unchanged, since their physical positions with respect to the source do not change.

To understand the difference between *Metric* and *Space* tuples, it is fundamental to focus on what happens when the source moves. In theory all the tuple instances must be changed because the origin of the coordinate system has shifted. This is exactly what happens in *MetricTuple* where the origin of the coordinated system is anchored to the source node. This

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

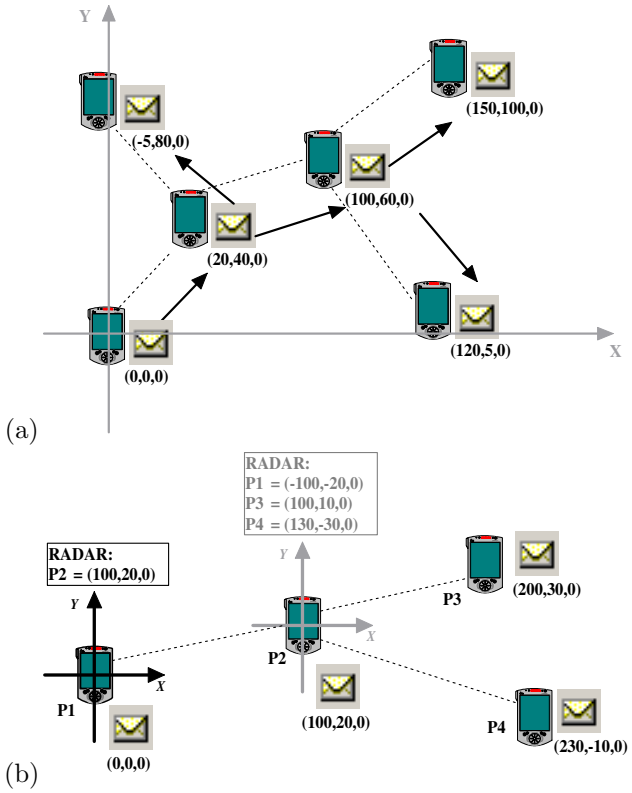


Figure 8. (a) *Metric* and *Space* tuples create a shared coordinate system, centered in the node that injected the tuple. (b) These tuples create the shared coordinate system, by having each node change the content of the tuples on the basis of the coordinates provided by the RADAR-like device.

of course can lead to scalability problems, especially if the source is highly mobile. What happens if also the source updates its value locally, without further propagating? In this case, the origin of the coordinate system remains where the tuple was first injected, even if no nodes are in that position. The coordinate system is maintained by the network, but not affected by it. This is the *SpaceTuple* implementation.

To further clarify the above concepts and to show again the expressiveness of our programming model, we are going to show another implementation of the QUERY field of the case study. This time the field will be implemented by means of a *QueryTuple* that holds the spatial distance from the source and propagates only within a limited physical scope. In the example, *QueryTuple* inherits from *MetricTuple* and so it represents the distance from the source, even when the source moves (see code in figure 9).

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

```
public class QueryTuple extends MetricTuple {
 public int value = 0;
 private int SCOPE = 10; // 10Km
 protected void changeTupleContent() {
  super.changeTupleContent();
  value = (int)Math.sqrt((x*x)+(y*y)+(z*z));
 }
 public boolean decidePropagate() {
  if(value < SCOPE) return true;
  else return false;
}}
```

Figure 9. A DistanceTuple is a tuple that holds the spatial distance from the source. Note that x,y,z are maintained in the *MetricTuple* and *SpaceTuple* classes

4. PERFORMANCE AND EXPERIMENTS

The effectiveness of the field-based approach in TOTA is related to costs and performance in managing self-maintained distributed tuples implementing fields: what is the cost of propagating a tuple? How much burden does self-maintenance add to the system? Is it scalable?

To start answering these questions, it is most important to assess whether the overhead on a node increases with the dimension of the network or not. If the answer to this question is negative, then the system is truly scalable: a node performs well independently of the size of the network in which it is embedded. If it is affirmative, then any implementation of the model is probably doomed to failure: performance degrades with an increase in network size.

The cost of propagating a tuple, relying on a multi-hop mechanism, is something inherently scalable. Each node will have to propagate the tuple only to its immediate neighbors. The size of the network does not matter since the global effort to spread the tuple is fairly partitioned between the constituting nodes.

The scalability of tuple maintenance is less clear. To be independent of the network size, maintenance operations must be confined within a locality from where something happened that changed the tuple structure (e.g. network topology change). If so, concurrent events (e.g. topology changes) happening at distant points of the network do not accumulate locally. If on the contrary maintenance operations always propagate across the whole network, distant concurrent events do accumulate and the system does not scale.

In this section we will present the results we found with regard to the different tuples in the hierarchy of figure 3. However, before proceeding, a caveat is needed: the following considerations and measurements refer only to the classes in hierarchy. It is clear that, by subclassing one of these tuples, a programmer can overturn the tuples' behaviors and possibly introduce complex cascading events that degrade performance.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

4.1. Structure and Message Tuple

These tuples are not maintained: so once propagated, they do not add any burden to the system. It is worth noting that also with regard to tuples that would add a custom reaction on the basis of local events (e.g. local sensor readings), maintenance operations scale. In fact, a change in the tuple context would affect only the tuple itself so it would be obviously confined (the *delete* operation in *MessageTuples* in an example of this situation).

4.2. HopTuples

With regard to *HopTuple*, answering our question is more complicated. Tuple maintenance operations are required upon a change in the network topology, to have the distributed tuples reflect the new network structure. This means that maintenance operations are triggered whenever, due to nodes' mobility or failures, new links in the network are created or removed. In this context our question becomes: are the tuple maintenance operations confined to an area neighboring the place in which the network topology had actually changed? This means that, if for example, a device breaks down (causing a change in the network topology) only neighboring devices should change their tuples' values.

How can we perform such localized maintenance operations in a fully distributed way? To fix ideas, let us consider the case of a *HopTuple* incrementing its integer content by one, at every hop, as it is propagated far away from its source.

Given a local instance of such a tuple X, we will call Y a *supporting tuple* of X if: Y belongs to the same distributed tuple as X (recall that TOTA marks all the tuples with an unique id), Y is one-hop distant from X, the value of Y is equal to the value of X minus 1. With such a definition, a supporting tuple of X is a tuple that could have created X during its propagation. Moreover, we will say that X is in a *safe-state* if it has a supporting tuple, or if it is in the node that first injected the tuple (i.e. hop value = 0). We will say that a tuple is not in a safe-state if the above condition does not apply (i.e. it has not any supporting tuple and it has hop value greater that 0). The basic idea is that a tuple that is not in a safe-state should not be there, since no neighbor tuples could have created it.

Each tuple should subscribe to the arrival or the removal of other tuples of its type in its one-hop neighborhood. Upon a removal, each tuple reacts by checking if it is still in a safe-state. In the case a tuple is in a safe-state, the removal has not any effect - see later. In the case a tuple is not in a safe state, it erases itself from the local tuple space. This eventually causes a cascading deletion of tuples until a safe-state tuple can be found, or all the tuples in that connected sub-network are deleted (as in the case of figure 5(b)). When a safe-state tuple observes a deletion in its neighborhood it can fill that gap, and reacts by propagating to that node. This is what happens in figure 5(a), safe-state tuple installed on node Ps and having value 3 propagates a tuple with value 4 to the hole left by tuple deletion (gray node). It is worth noting that this mechanism is the same as when a new peer is connected to the network. Similar considerations apply with regard to tuples' arrival: when a tuple senses the arrival of a tuple having a value lower than its supporting tuple, it means that, because of node mobility, a short-cut leading to the source has been created.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

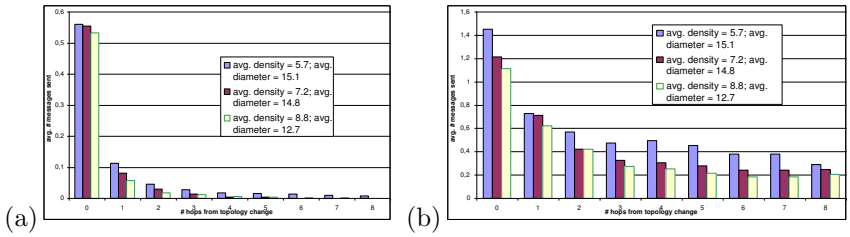


Figure 10. Experimental results: locality scopes in tuple's maintenance operations emerge in a network without predefined boundaries. (a) topology changes are caused by random peer movements. (b) topology changes are caused by the movement of the source peer.

How much information must be sent to maintain a distributed tuple? What is the impact of a local change in the network topology in real scenarios? To answer these questions we exploited a mobile ad-hoc network emulator running the TOTA middleware, that we developed. The graph in figure 10 shows results obtained by a large number of experiments, conducted on different networks. We considered networks having an average density (i.e. average number of nodes directly connected to an other node) of 5.7, 7.2 and 8.8 respectively (these numbers come from the fact that in our experiments they correspond to networks composed by 150, 200, 250 peers over the same area - the density increases because peers are more packed). In each network, a *HopTuple* was propagated. Nodes in the network move randomly, continuously changing the network topology. The number of messages sent between peers to keep the tuple coherent was recorded. Figure 10(a) shows the average number of messages sent by peers located in an x-hop radius from the origin of the topology change. Figure 10(b) shows the same values, but in these experiment only the source of the tuple moves, changing the topology.

The most important consideration we can make looking at those graphs, is that, upon a topology change, a lot of update operations will be required near the source of the topology change, while only few operations will be required far away from it. This implies that, even if the TOTA network and the tuples being propagated have no predefined boundaries, the operations to keep the tuples consistent are strictly confined within a locality scope (see figure 10). This fact supports the feasibility of our approach in terms of its scalability. In fact, this means that, even in a large dynamic network with a lot of nodes and tuples, the self-maintenance of tuples does not have to continuously flood the whole network with updates, eventually generated by changes in distant areas of the network. Updates tend to be confined within the locality scope from where they took place.

4.3. Metric and Space Tuple

With regard to these tuples, determining if their self-maintenance operations are confined is rather easy. We said that a *Metric* tuple's maintenance is confined to the node itself for all the nodes apart from the source, it spreads across the whole network if the source moves.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

So the answer for *Metric* tuples is partially negative and for this reason they must be used carefully, maybe with custom rules in their propagation rules limiting a-priori their scope, or by triggering update operations only if the source node moves by a certain amount (e.g. trigger update only if the source moves by at least 1m). The answer for a *Space* tuple, instead, is clearly affirmative, since maintenance is strictly locally confined.

All the above considerations are good hints for the feasibility of the model, showing that it can scale to different application scenarios.

5. CONCLUSION AND FUTURE WORK

In this paper we have presented a programming model to create self-maintained distributed tuples over dynamic networks. These kinds of tuples are, in our opinion, a fundamental building block in the context of field-based coordination. However, several issues are still to be investigated to completely assess the effectiveness of our model. First, more applications should be developed to verify the completeness of our abstractions. Second, more experiments and performance evaluations are needed to test the limits of usability and the scalability of TOTA. Third, an underlying general methodology, enabling engineers to map a specific coordination policy into the corresponding definition of tuples, should be defined. Finally, we must integrate proper access control models to govern accesses to distributed tuples and their updates.

REFERENCES

1. E. Bonabeau, M. Dorigo, G. Theraulaz, *Swarm Intelligence*, Oxford University Press, Oxford (UK) 1999.
2. G. Cabri, L. Leonardi, M. Mamei, F. Zambonelli, *Location-dependent Services for Mobile Users*, IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems And Humans, Vol. 33, No. 6, pp. 667-681, November 2003.
3. O. Khatib, *Real-Time Obstacle Avoidance for Manipulators and Mobile Robots*, Intl J. Robotics Research, vol. 5, no. 1, 1986, pp. 9098.
4. S. Johansson, A. Saffiotti, *Using the Electric Field Approach in the RoboCup Domain*, RoboCup 2001, Seattle (WA), 2001.
5. M. Mamei, L. Leonardi, F. Zambonelli, *Co-Fields: A Unifying Approach to Swarm Intelligence*, 3rd International Workshop on Engineering Societies in the Agents World, Madrid (Sp), 2003.
6. M. Mamei, F. Zambonelli, L. Leonardi, *Cofields: An Approach to Distributed Motion Coordination*, IEEE Pervasive Computing, 3(2):52-61, 2004.
7. M. Mamei, Franco Zambonelli, *Programming Pervasive and Mobile Computing Applications with the TOTA Middleware*, 2nd IEEE International Conference on Pervasive Computing and Communication (Percom2004), Orlando (FL), 2004.
8. O. Babaoglu, H. Meling, A. Montresor, *A Framework for the Development of Agent-Based Peer-to-Peer Systems*, 22nd International Conference on Distributed Computing Systems, Vienna (A), 2002.
9. R. Menezes, R. Tolksdorf, *A New Approach to Scalable Linda-systems Based on Swarms*, ACM Symposium on Applied Computer 2003, Orlando (FL), 2003.
10. R. Nagpal, *Programmable Self-Assembly Using Biologically-Inspired Multiagent Control*, 1st International Conference on Autonomous Agents and Multiagent Systems, Bologna (I), 2002.
11. R. Poor, *Embedded Networks: Pervasive, Low-Power, Wireless Connectivity*, PhD Thesis, MIT, 2001.
12. W. Shen, B. Salemi, P. Will, *Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots*, IEEE Trans. on Robotics and Automation 18(5):1-12, 2002.
13. S. Johnson, *Wild Things*, Wired, 10:03, March, 2002.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls